Exam
DT500A - Distributed Systems

2nd of January 2020

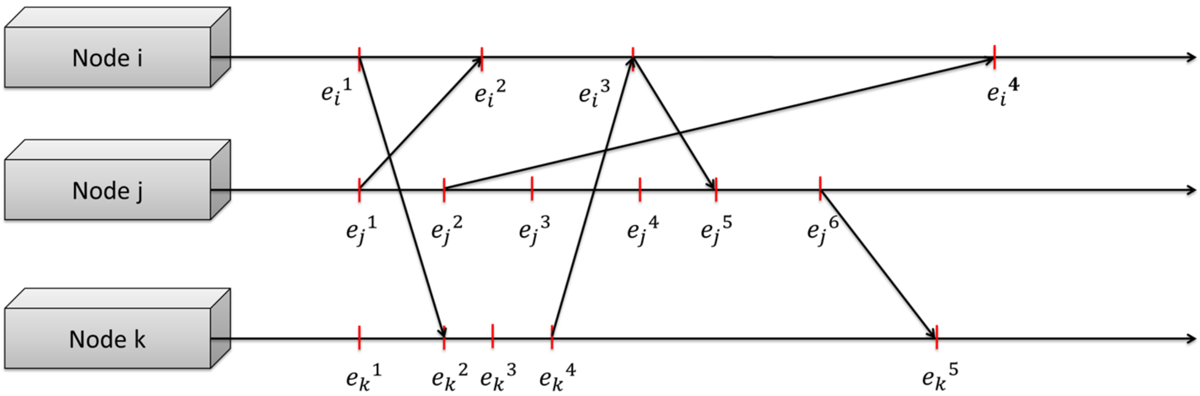
Responsible teacher: Erik Schaffernicht

Aids:	No additional aids besides the task sheets (which include the “cheat sheet”).
Questions:	Erik can be reached during the exam via phone (019 30) 3227.
Points:	Each task has its point value listed, indicating how extensive the answer is expected to be.
Comments:	**Read the tasks carefully** and answer what the task asks for. Write legibly. Extended bullet points are fine as long as they clearly contain the arguments. There is no need to write an essay. There is neither partial nor total order in which the tasks have to be answered, but clearly indicate which answer refers to which task.

Task 1 – Consistency (2 points)

What kind of consistency model would you use to implement an electronic stock market? Explain your choice.
The replicated data to consider consists of the current buy and sell orders for a limited number of items/stocks, which are replicated for availability (scaling with user numbers).

Task 2 – Logical clocks (4 points)



Use logical clocks and the synchronization method suggested by Lamport to establish a *partial potential causal order (ppco)* for the events and messages (as indicated by arrows) shown above. Below are given two pairs of events A & B. For each of them decide whether *ppco*(A)<*ppco*(B), *ppco*(A)>*ppco*(B) or *ppco*(A)=*ppco*(B), and A↪B, B↪A or A?B (to denote concurrent events in the "happens before" relationship ↪) is correct.

i) e_k^3 & e_j^4

ii) e_j^5 & e_k^3

Task 3 – Failure tolerance of election algorithms (2 points)

Given a distributed system with 10 nodes with no current coordinator, how many rounds do the algorithms Floodset and Bully need to run in order to determine a new coordinator in the presences of at most

(a) 3 node failures and no messages are lost

(b) 1 node failure and at most 2 lost messages?

Task 4 – The coffee machine (14 points)

There is a coffee machine, customers who buy coffee at the machine, and a supplier that refills the coffee machine.

- The machine can handle only one task at a time: giving out one coffee or being refilled by the supplier. The end of each task is announced by the machine to the client or supplier.
- After accepting a coffee buy order, the client has to provide payment for the coffee on request of the machine.
- During the execution of a task the machine places coffee orders in a queue. After finishing its current task, the machine will execute the next request in a first-come-first-serve order.
- If the machine is empty it informs the supplier about the need for a refill and will not serve anymore coffee to clients.
- The supplier can refill a machine that is not empty.

Assumption: There are **no** failures of any kind happening in this scenario.

Task 4 – The coffee machine contd. (14 points)

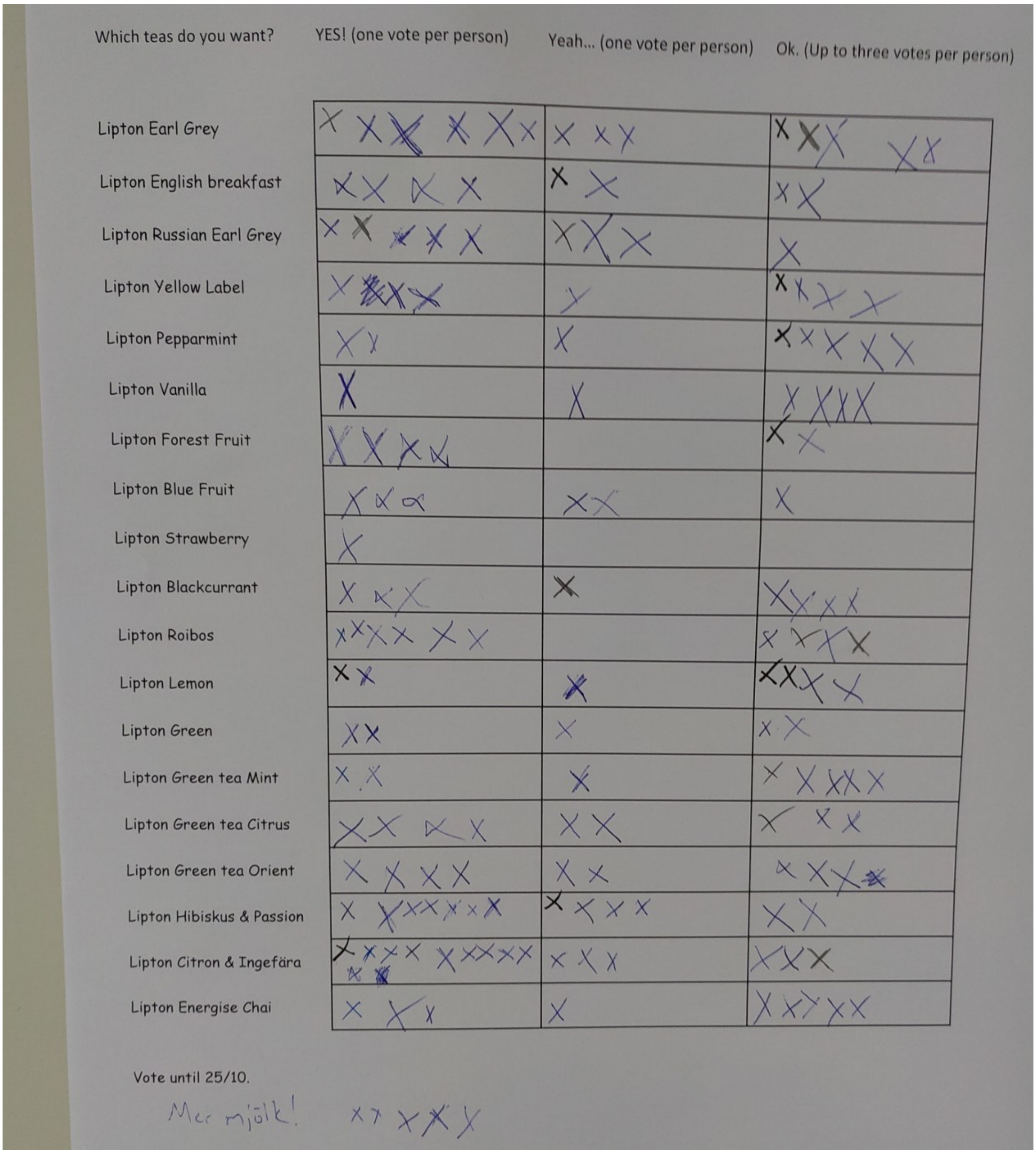
A) Write pseudocode for the supplier, customers and the machine, modeling the above behavior through exchanging messages between nodes. Clearly indicate the use of asynchronous and synchronous messages. (7 points)

B) Using your solution, draw a sequence diagram for the following situation: Customer A orders and pays a coffee, during execution customer B is ordering a coffee, but after serving the coffee for A the machine is empty. The diagram ends when customer B receives their coffee. (4 points)

C) Explain three different problems that can occur, if the assumption that there are no failures is no longer valid! (3 points)

Task 5 – Consensus (4 points)

Below you will find a picture with an informal voting algorithm applied for finding consensus within a known group of nodes. What kind of communication model is used here? Name three problems that the suggested method has and suggest improvements.



Communication Models (Lecture 3) Message-based Communication (Basics)
Base for other communication models. There are two roles: sender & receiver. Messages sent between. Example: Socket messaging
One or more receivers (unicast/multicast), failures: reliable/unreliable send, termination *semantics*: synch. or asynch. send/receive
Indirect communication via ports. Unreliable send: Sender don't know if message sent. Reliable send: Sender recieves ACK when message sent. Synchronous: Sender waits for receive message until next action, DoS attack by not answering. Asynchronous: Sender don't wait until ack received, DoS attack by flooding. Same with sync recv. and async recv. Fail-stop: nodes stop sending messages for ever. Fail-stop-return: nodes stop sending messages for a while, or the messages are heavily delayed. Byzantine failure: the messages from nodes might be lost, delayed or changed. **Failure semantics** Exactly-once: one and only one message is sent or received. At-least-once: one or more messages sent or received (make sure at least one message is sent). At-most-once: zero or one message delivered/received (may or may not be sent/received).
Client/Server: Classic client & server model, client(s) sends packets to the server and it may respond.
RPC: client sends procedure to server, e.g client sends procedure to delete file from server, a stub is being used by both parties as a proxy to ensure client signature. The goal is to make it seem as if the remote calls are done as local calls, ie I open a file on that is located on a remote server, the OS will do a RPC to open a file on a remote server which will respond with the result. It will look like it is located on my computer… Messages are synchronous, all nodes know each other.
Blackboards: Users put tuples with data on a blackboard (client sends variables with data to server) and other users can request one or more messages when they need it. Specialist = reads/adds messages to blackboard, moderator = makes sure old messages are removed and that the messages are legit. This model is open and dynamic (a specialist can join/leave whenever it wants in the process), anonymous for specialists (specialist do not need to know other specialists). But blackboard and moderators are bottlenecks as well as single points of failures.
Event-based: Application: Trading systems, news agencies, surveillance systems, flexible replication systems. Publisher: Registers subscribers(subs.) & their interests, sends message to subs in case of events, advertise offered event repertoire. Subscriber: Subscribes to event messages from publisher (pub.), receives event message. Pros: Openness, interoperability, size scalability, IT security, reusability, timeliness(major improvement compared to Blackboard) Cons: Efficiency, storing events, availability, design & implementation.
Pipelines/stream-based: Goal: Communicating high volume of data streams in real time(video streams). Failure and termination semant.: High data volume -> efficiency is important -> no universal failure model. Problems: All problems gets solved with synchronization.
Transactional memories: What is transactions?: Collection of operations that adhere to ACID. ACID: Atomicity: transactions don't depend on each other, Consistency: transaction start and end in a consistent state, Isolation: concurrent transactions do not influence each other, Durability: Result of finished transaction is permanent
Naming(Lecture 4) three naming system(Flat, structured, attribute-based naming**) Flat:** Name is a random string and does not contain any information on how to locate the process. how flat names can be resolved, or, equivalently, how we can locate an entity when given only its identifier. Forwarding pointers **idea** When an entity moves, it leaves behind a pointer to its next location and update a client's reference when present location is found. Geographical scalability **problems**: Long chains are not fault tolerant, Increased network latency at dereferencing. Hierarchical Location Service **Idea** Large-scale search tree dividing the network into hierarchical domains. Size Scalability **Problem:** root node needs to keep track of all identifiers.
Structured: A namespace is a labelled, directed graph consisting of leaf nodes and directory nodes. **Leaf node** represents a named entity, e.g. by storing its address **Directory node** An entity that refers to other nodes. **Problem** To resolve a name, we need to find and start at a directory node. Two methods Iterative name resolution and Recursive name resolution (Datakom DNS lookup). **Size scalability** need to ensure that servers can handle a large number of requests per time unit→ high-level servers are in big trouble. **Solution**: Replication Top-level server is heavily replicated → start at the closest server, Node content on the top-level rarely changes. Mounting **Idea** Merge different name spaces transparently by connecting the node identifier of a foreign name space with a node in the current name space **Required Information** Used access protocol, Server name, Mounting point in the foreign name space.
Attribute: Idea Naming and look-up of entities by means of their attributes→ Directory services. **Problem** Comparison of requested and actual attributes is expensive→ Inspection of all entities required. **Solution** Combination of structured naming with a database implementing the directory service
Time & sequences (Lecture 5): Computer clocks: electrical component that counts the oscillation of quartz crystals, associated are 2 registers, a counter & a holding register. Decrements counter by 1, interrupt generated when counter gets to 0 & then reloaded from the holding register**. Sync. of phys. Clocks** - lvl of correspondence between a comp. clock and UTC depends on 1.precision of the reference clock 2.jitter of the signals: distance from the reference, atmospheric conditions 3.drift of local clocks 4. Synch. algorithm. **Quantifying drift:** synch. so the clock deviation is less than α. (t : reference time(UTC), C(t): value of the comp. clock at time t) Ideally: $C(t)= t$; $C(t2)-C(t1)/(t2-t1) = dc/dt = 1$. **P maximum drift rate**, part of clock specification - clock working within specification iff 1-p<-dc/dt<-1 + p. Resynchronize clocks with a max deviation α every $\Delta t = \alpha$ /(p1+p2) sec. **Offset estimation** o=(t3-t4+t2-t1)/2 +(treply-treq)/2 (treq & reply are the unknows, if term too big, dont synch and try again instead). **Precision:** divergence from UTC in WANs up to 50msec & up to 1msec in LANs, **reliability/security:** fail-stop of a stratum-i server→ dependent stratum-i+1 servers need to connect to a new stratum-i server, digitally signed time info→ authenticity&integrity, **scalability**: load distribution via distributed server system. **Use case:RPC failure semantics**:goals -avoid management costs for RPC call IDs -define a time for which results are stored on the server. **Approach** -RPC client: each call contains -unique client ID(not for each individual call) -originall call time stamp. RPC server: stores last time stamp per client non-persisently -storage of results of calls -duplicate filtering and resending of results. **Fail-stop-return scenario→** server not recognizing duplicates anymore after restart. **Idea**: server knows a rough time estimate of failure persistenly -recognizes all RPCs that are potential duplicate calls. **Required:** timestamp of the RPC -failure time. **Consequence:** -a not executed RPC will be considered a duplicate. -no harm done: at-most-once compliant, handling of "lost" RPCs is then task of client. **Prerequisites**: client/server share a synch time with a max divergence -client attached to each RPC its birthdate -global constant for the max life span of RPCs. → server computes a cut-off time L(lates). (L=Tserver -MLS -divergence). **Algorithm** -every ∆t time units the server writes Tserver into persistent memory -at restart: L← stored.Tserver +∆t (RPCs arriving with younger time stamp(alive), with older(dead, either executed or badluck). L solves problem regarding storage time of results. **Casual past**{f|f→e}, **casual future**{f|e→ f}, **casual independence** $\neg(e\to f) \wedge \neg(f\to e) \Rightarrow$ e and f are causal independent. **Lamport's algorithm**, - • each node ni uses a local clock lci • $ppco(ei)= lci$, each event in ni is timestamped by lci : ei^lci • lci is added to each message sent by ni. **Total potential causal order•** I is a finite index set with a total strict order over "<" • all nodes ni have a unique identifier $i \in I$ globally unique ID • lci is the synchronized local logical clock of ni. **Partial causal order•** each node ni uses a local vector clock vci • $pco(ei) = vci$, each event in ni is timestamped by vci : ei^vci • vci is added to each message sent by ni.

Lecture 6 (Coordination) Central coordination(Adaption of the methods of non-distributed systems): //**1,** All competitors agree on a coordinator. **2,** Before entering a critical section**;2a,**sending in application to the coordinator(**enter**). **2b,** waiting for the go-ahead from the coordinator (**grant**). **3,** on leaving a CS informing the coordinator (**leave**).//The coordinator uses a First-Come-First-Serve priority-based queue.
Decentralized mutual exclusion(removal of the coordinator as bottleneck and single-point-of-failure): Direct communication between all competitors -> **enter** message to everyone. Response to **enter** is; **grant**, if there is no conflict. Otherwise conflict handling follow a strategy (FCFS). **Requirements;** All competitors know each other. Distributed FCFS requires total order of events(tpco or tco). Bookkeeping in each client for each critical section **C**; **C.state**{**free**, **occupied**, **requested**}, **C.time** - timestamp of enter, **C.rivals** - set of competitors waiting for **grant**, **counter** - for the number of received grants for enter. **//1.** Requesting a CS; **1a, C.state** <- requested. **1b, C.time** <- tpco. **1c.** Sending **enter** message to all other nodes containing the requested CS, the node's ID and the node's time. **2,** Entering a CS; **2a,** wait for **n-1** grant messages. **2b, C.state** <- **occupied**. **3.** Leaving a critical section; **3a, C.state** <- **free**. **3b**, Sending **grant** message to all nodes in **C.rivals**. **4.** Response to **enter** message from node **ni** at time **t**; **4a,** If **C.state** is == **free** -> respond with a **grant** message to **ni**. **4b,** If **C.state** == **occupied** -> requesting node is added to **C.rivals**. **4c,** If **C.state** == **requested** & **C.time** '-> **t** ->node is added to **C.rivals**. **4d,** If **C.state** == **requested** & **t** '-> **C.time** -> **grant** message to **ni//**
Token ring algorithm(Overlay network with ring topology. For each CS there is a token moving in the ring. Entering the CS is only possible if the node is in possession of the token): <u>Node behaviour;</u> On receiving a token either enter the CS or pass the token along to the next node in the ring. On leaving the CS the token is given to the next node
Election algorithm(Instead of using a predefined coordinator the system elects a coordinator and reelects a new one in case of a fail-stop of the current coordinator): <u>Starting condition;</u> Either there is no coordinator, one node detects a failure of the current coordinator or a failed coordinator comes back to life)
Bully algorithm(The most important node alive will get the coordinator role, by gradually telling everyone else who wants to role that they are not getting it. Once everyone else has quit, the new coordinator will inform the other nodes of its victory)<u>Starting condition:</u> Same as election algorithm. <u>Algorithm(1);</u>**//1.** Node **ni** initiates election by sending **elect** to all nodes in **nj** with **j>i**. **2**, Node **nj** receives **elect**; **2a**, **reply** to the sender. **2b**, starts an election on its own**//.** <u>Algorithm(2);</u>//<u>Node ni:</u> **1**. Receives at least one **reply**; **1a**, There is an active node with a higher ID -> no further actions. **2**, Receives no answer; **2a**, There is no active node with higher ID -> ni becomes coordinator. **2b**, inform everyone ->**bully(ni)** to all nodes. <u>Node nk:</u> **1** Receives **bully**; **1a** update **ci <- ni**.**//**
Changs ring algorithm(Overlay network with ring topology. Determine maximum over node IDs. Publication of the maximum ID): <u>Requirements:</u> Finite index set with total order. At least partially known ring topology in a way to determine successors. <u>Algorithm</u>: **//1,** Node ***ni*** initiates election by sending **elect(i)** to successor node. **2,** Node nj receives elect(i); **2a,** If **i != j**: **nj** sends **elect(max(i,j))** to successor node. **2b,** If **i = j**: **nj** sends **meCoordinator(i)** to successor node. **3**, Node **nj** receives **meCoordinator(i)**. **3a**, If **i != j**: update **cj <- ni** and forward message to successor. **3b**, If **i = j**: finished**//**
Floodset algorithms(Building a local repository of all voting information. Distribution of all the local knowledge in multiple communication rounds. Maximum redundancy): <u>Algorithm:</u> **//1, Round 1:** Each node broadcasts their own vote. **2, Round 2 to round f+1:** Broadcast all known values. **3, End of round f+1:** Compute consensus results**//**
Exponential information gathering: <u>Requirements:</u> Finite index set with total order -> each node has a unique ID. Each node knows all other nodes. Number of failures has a known upper bound **f**. **Authentic** messages, but not necessary of **Integrity**. <u>Algorithm:</u> **//1,** Each node constructs iteratively a local EIG tree **T**; **1a,** The nodes own vote is the root of the tree. **1b,** Round **k**, **k>0**; **1b1,** Layer **k-1** is sent to all other nodes (flooding). **1b2,** Layer **k** is constructed with the received messages from other nodes. **2,** With upper limits of failures **f**, the number of rounds is **f+1**. **3,** After **f+1** rounds each node has EIG tree of depth **f+1** to determine voting results**//**
Byzantine failures - a consensus algorithm. <u>Three steps;</u> 1, Exchange in opinions - every general sends their vote to everyone else. 2, Validation - All generals send the information from step 1 to everyone else. 3, Decision - Each general decides based on the received information.
Replication & Consistency (Lecture 7)
Consistency model: <u>Strict consistency:</u> Any read on data item returns a value corresponding to the result of the most recent write on that data item. <u>Sequential consistency:</u> The order of operations applied on each replica is the same. Operations from the same node follow the order given by its program. <u>Causal consistency:</u> The order of potentially causally related write operations applied on each replica is the same. Concurrent write operations can have different order in each replica. <u>FIFO consistency:</u> Write operations from a single node are applied to each replica in the correct order, but writes from different nodes may be applied to each replica in a different order.
Reading and writing from/to nodes: <u>Monotonic Reads:</u> Reading the value of a data item from a specific node, any successive read operation of that item by that node will always return that same or a more recent value. <u>Monotonic Writes:</u> Writing the value of a data item from a specific node will be completed before any successive write operations from the same node. <u>Read-your-writes consistency:</u> The result of writing a value of a data item from a specific node will always be seen by successive read operations by the same node. <u>Write-follows-Read consistency:</u> A write operation of a value of a data item from a specific node following a previous read by the same node, is guaranteed to take place on the same or more recent value of the data item.
Three different types of replicas: <u>Permanent replicas</u> node always having a replica. <u>Server-initiated replica</u> node that can dynamically host a replica on request of another server in the data store. <u>Client-initiated replica</u> node that can dynamically host a replica on request of a client.
Consistency protocols: <u>Primary-based protocols</u>: Purpose: Implementing sequential consistency. Idea: One replica acts as coordinator (primary) for all updates to a certain store. <u>Replicated-write protocols:</u> Idea: Each read or write operation requires permission by a number (quorum) of replicas before execution, subject to the following constraints: $N_R+N_W>N$ and $N_W>N/2$, where N: number of nodes/replicas, NR: number of nodes necessary to contact for *read,* NW: number of nodes necessary to contact for *write*. <u>Cache-coherence protocols:</u> Combination of previously discussed protocols (often primary-based) and results from computer architectures dealing with • coherence detection • coherence enforcement.
Security aspects in distributed systems (Lecture 8)
<u>Security policies</u> - Cryptography: Symmetric encryption uses a single secret key for both encryption and decryption while asymmetric encryption uses one public key for encryption and one secret for decryption**, Secure channels:** User login authentication, authentication of communicating entities, hashing of data to ensure integrity **, Access control:** Access control lists known from OS filesystems, Firewalls provide packet filtering (based on source and destination address in the packet header, network layer) and application gateway(looks at the content of incoming and outgoing messages, application layer)**, Security management:** Responsible for crypt key management/exchange for symmetric cryptography and acts as key distribution centers for asymmetric cryptography.